INTRODUCTION TO IMAGE PROCESSING AND COMPUTER VISION

Project 2: Surface Crack Detection

Emilia Wróblewska

January 24, 2021

Contents

1 Introduction

Image Classification is a process of classification based on various information from given set of pictures. It utilizes pattern recognition in computer vision and machine learning models in order to learn how to assign an image into its respective class. Image classification is perhaps the most important part of digital image analysis and has countless applications, such as face recognition or automated photo organization.
The goal of this task was to create a machine learning model which for given feature descriptors would be able to assess whether a given image of some surface contains a crack (positive) or not (negative). The data set is really large, it consists of 20 000 images per class - so a total of 40 000 pictures. All of them have a fixed size of 227x227 pixels which is very convenient since there is no need for equalizing their areas.

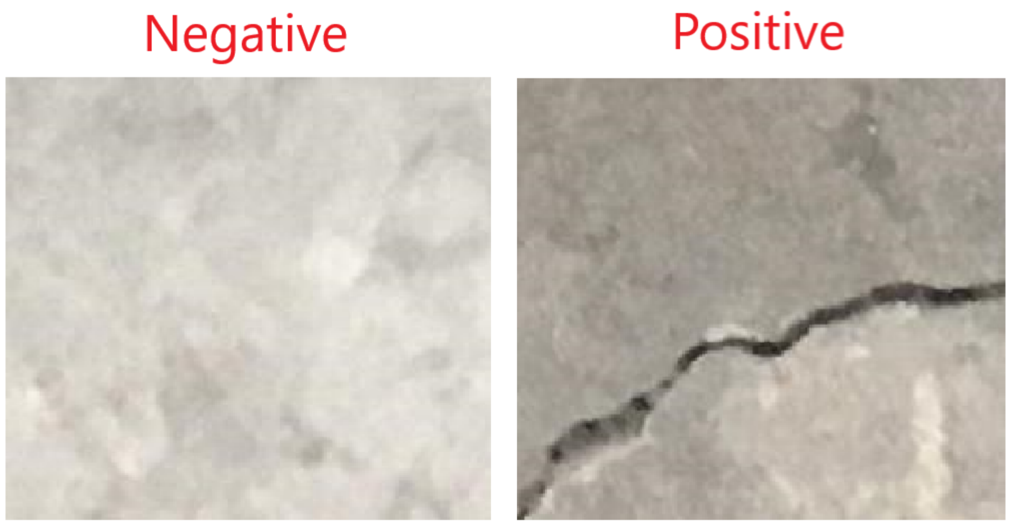


Figure 1: Example of classification for 2 images

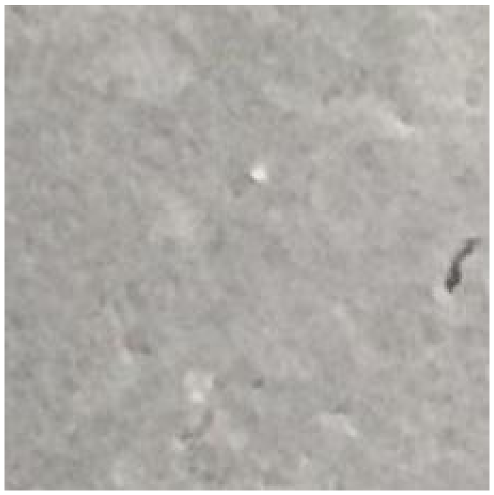
2 Algorithm Description

The algorithm is divided into 2 parts: extracting feature descriptors and training the model. Since the problem is quite simple (there are only 2 classes) and the data set is more than sufficient, the main idea was to test various combinations of descriptors and compare their prediction scores.

Another approach was to assess the impact of training set size on the final accuracy of prediction. This required the model to be trained on increasing parts of original data set, starting from 1000 images per class. Since this part doesn't introduce any new algorithm into the solution, the results will be discussed in the last section.

2.1 Feature Descriptors

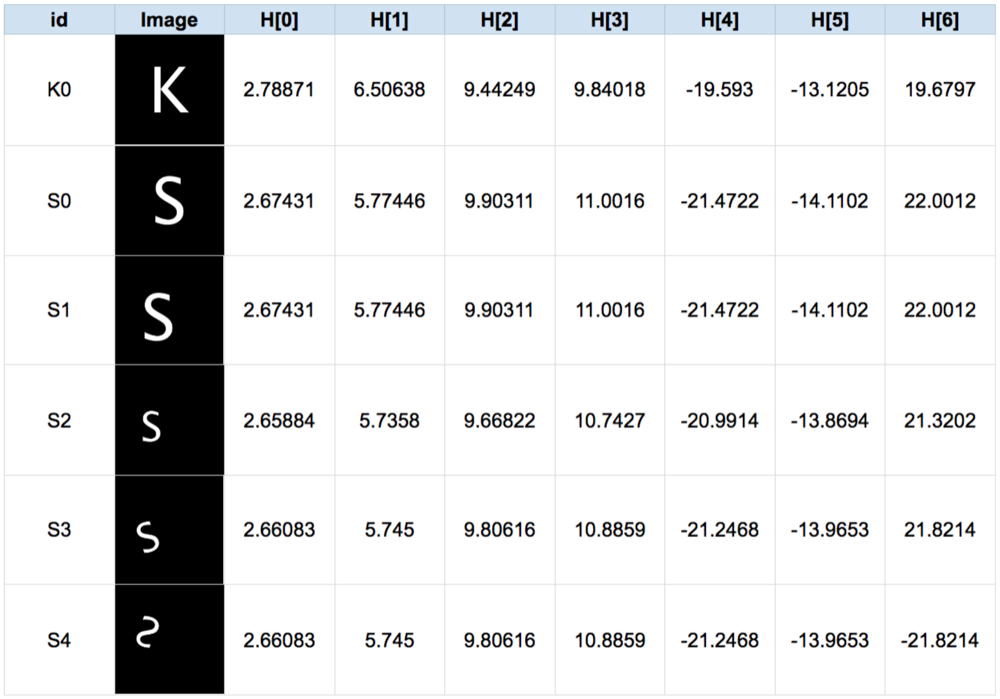
In total 6 distinct feature descriptors were used. First 3 of them are global features, and the remaining 3 are local features. The following exemplary visualizations of different features will be shown for the first two images from each class i.e. photos 00001.jpg from 'Positive' and 00001.jpg from 'Negative' folder.


(a) Negative 00001.jpg

(b) Positive 00001.jpg

1) Hu Moments

Image moments are a weighted average of image pixel intensities. They capture information about the shape of a blob in a binary image because they contain information about the intensity I(x,y), as well as position x and y of the pixels. Hu Moments are a set of 7 numbers calculated using central moments that are invariant to image transformations. The first 6 moments have been proved to be invariant to translation, scale, and rotation, and reflection, while the 7th moment's sign changes for image reflection.[1]



id	Image	H[0]	H[1]	H[2]	H[3]	H[4]	H[5]	H[6]
K0	K	2.78871	6.50638	9.44249	9.84018	-19.593	-13.1205	19.6797
S0	S	2.67431	5.77446	9.90311	11.0016	-21.4722	-14.1102	22.0012
S1	S	2.67431	5.77446	9.90311	11.0016	-21.4722	-14.1102	22.0012
S2	S	2.65884	5.7358	9.66822	10.7427	-20.9914	-13.8694	21.3202
S3	S	2.66083	5.745	9.80616	10.8859	-21.2468	-13.9653	21.8214
S4	S	2.66083	5.745	9.80616	10.8859	-21.2468	-13.9653	-21.8214

Figure 2: Example of Hu Moments for different photos

Image	H[0]	H[1]	H[2]	H[3]	H[4]	H[5]	H[6]
Negative	$9.89 \cdot 10^{-4}$	$6.19 \cdot 10^{-12}$	$2.26 \cdot 10^{-13}$	$1.08 \cdot 10^{-13}$	$5.16 \cdot 10^{-27}$	$-2.08 \cdot 10^{-19}$	$1.61 \cdot 10^{-26}$
Positive	$1.28 \cdot 10^{-3}$	$4.91 \cdot 10^{-10}$	$3.25 \cdot 10^{-14}$	$8.85 \cdot 10^{-14}$	$1.55 \cdot 10^{-27}$	$6.21 \cdot 10^{-19}$	$-4.485 \cdot 10^{-27}$

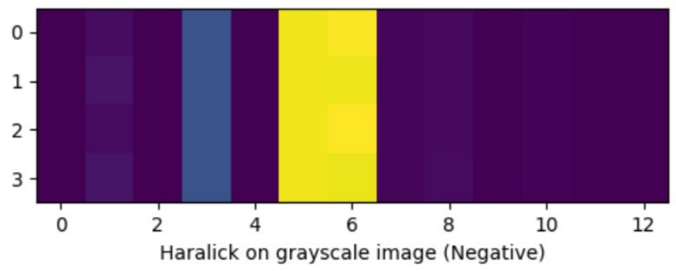
Table 1: Hu Moments for Negative and Positive images

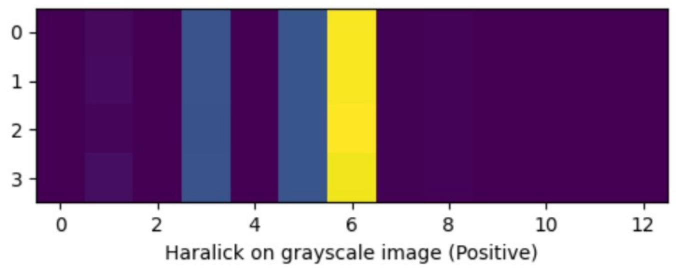
Due to the fact that generally cracks have rather vague shapes, Hu Moments may have problems at detecting them. As for the Python algorithm, it is really easy. Firstly, the image is converted into grayscale and then, Hu Moments are extracted using OpenCV functions. The result is 'flattened' to obtain a one-dimensional vector describing the image.

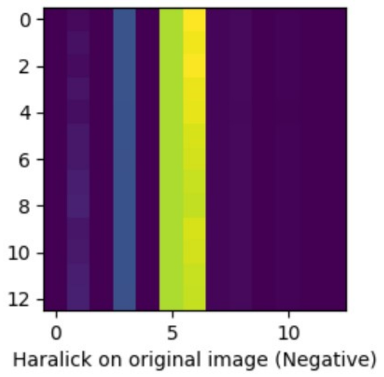
```python
# 1: Hu Moments
def fd_hu_moments(image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    hu = cv2.HuMoments(cv2.moments(image)).flatten()
    return hu
```

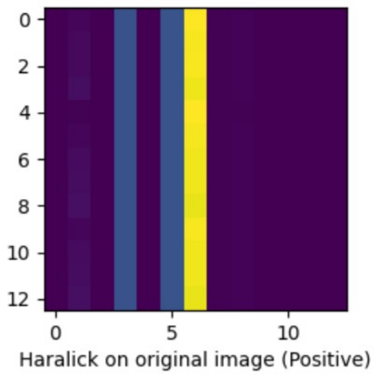
2) Haralick Texture

Image texture provides information about the spatial arrangement of color or intensities in an image or its selected regions. In other words, it can be represented as set of metrics quantifying the perceived texture of given picture. Haralick texture is calculated from a gray level co-occurance matrix (GLCM) and it is a most common technique of obtaining texture descriptors. The following figure shows the Haralick feature extracted from our 2 exemplary photos.







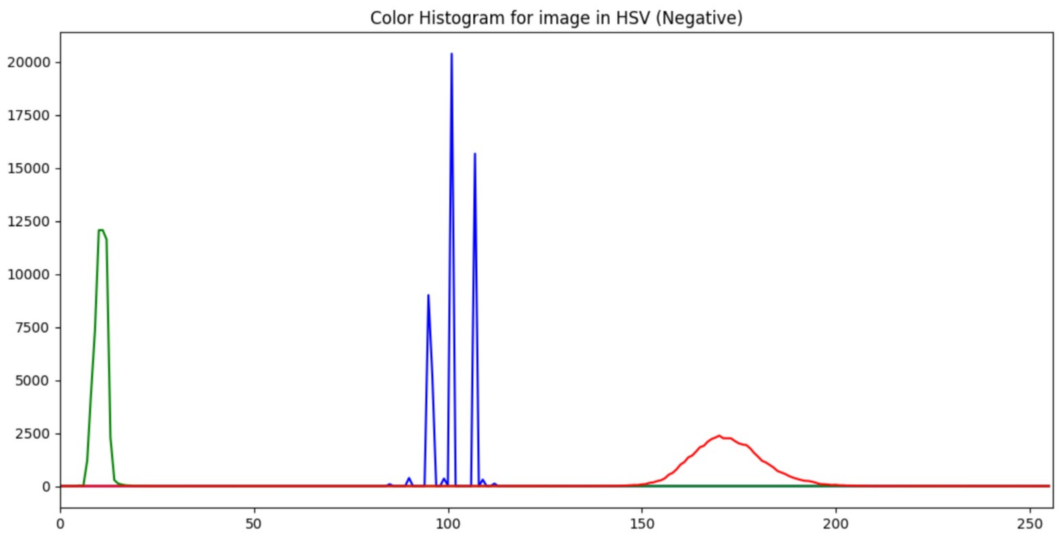


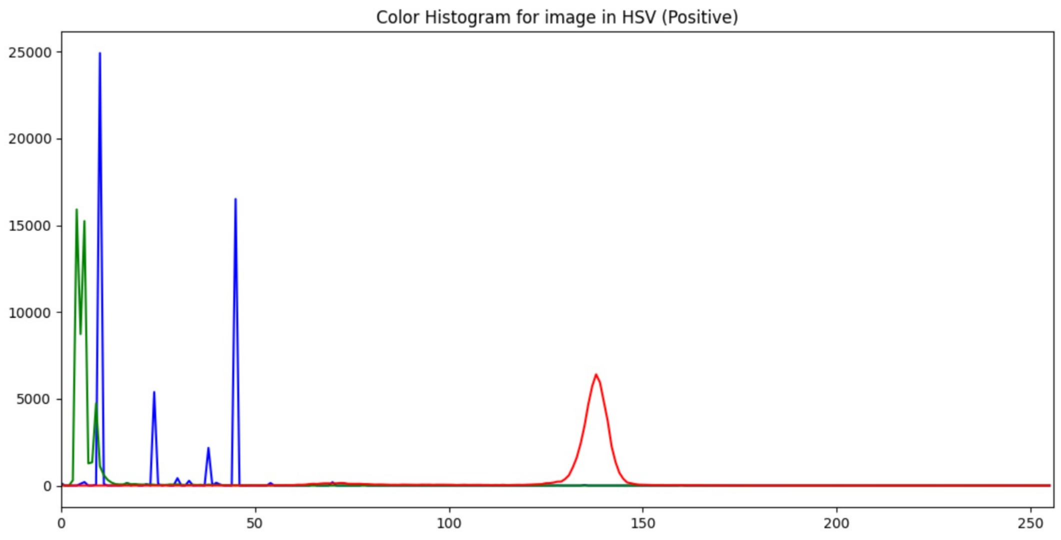
To extract Haralick texture we use Python mahotas library and again convert the result into a single descriptor vector.

```
# 2: Haralick Texture
def fd_haralick(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    haralick = mahotas.features.haralick(gray,
                        ignore_zeros=True).mean(axis=0)
    return haralick
```

3) Color Histogram

In image processing and photography, a color histogram represents the distribution of colors in an image and is calculated separately for each color channel. For digital images, a color histogram represents the number of pixels with colors belonging to one of specific ranges spanning the image's color space. In this case the histogram was built for three-dimentional HSV space due to it's slightly higher accuracy score than for BGR color space. The final descriptor is obtained by normalizing and 'flattening' the histogram.





```
# 3: Color Histogram
def fd_histogram(image, mask=None):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    hist = cv2.calcHist([image], [0, 1, 2], None,
            [bins, bins, bins], [0, 256, 0, 256, 0, 256])
    cv2.normalize(hist, hist)
    return hist.flatten()
```

4) Local Binary Pattern (LBP)

Local binary patterns (LBPs) are a more recent set of features where each pixel is looked at individually and its neighbourhood is analysed and summarised by computing a single value. The final descriptor vector is a normalized histogram across all the pixels in the image. LBP is considered as a powerful feature for texture classification, because it is insensitive to orientation and illumination of the image.

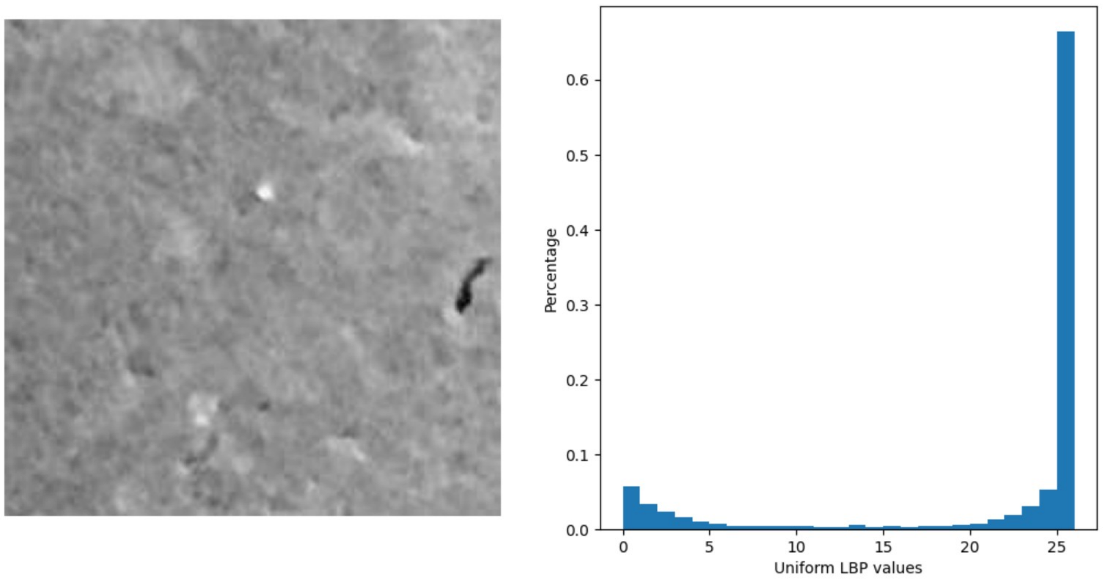


Figure 3: LBP Histogram of Negative image

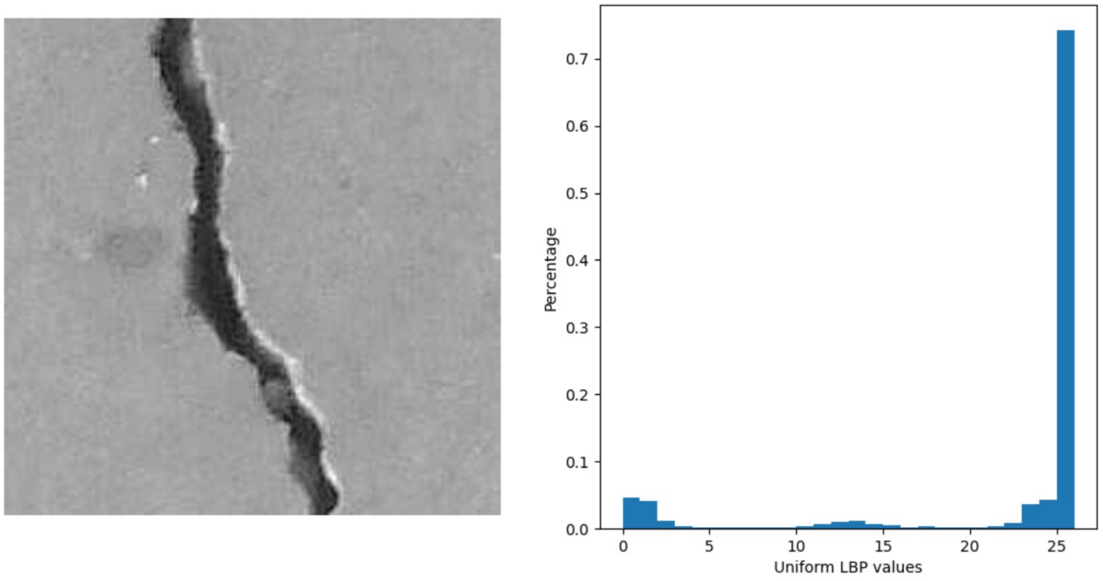


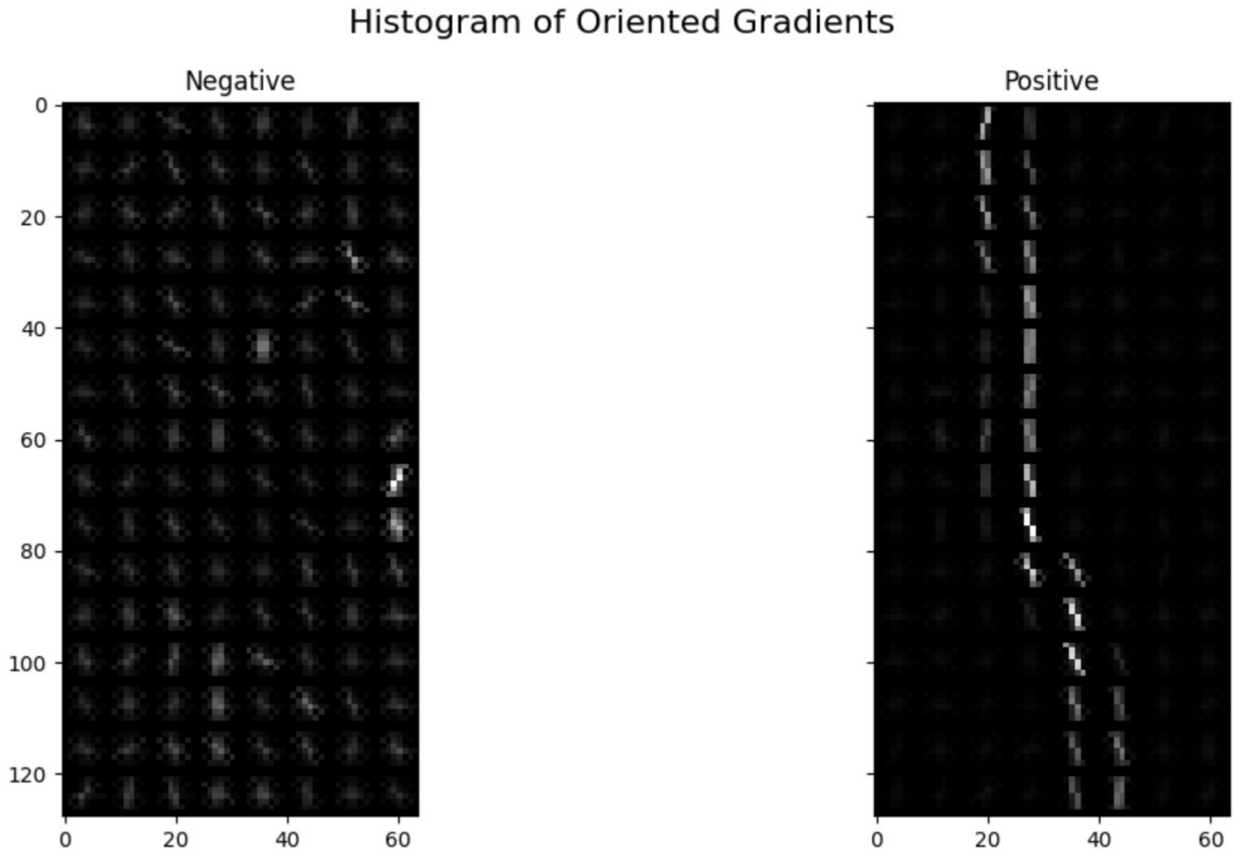
Figure 4: LBP Histogram of Positive image

The implementation of LBP is slightly more complex, due to the amount of options and variables needed for good description of pixels' neighbourhood, like the number of circularly surrounding points for each pixel, radius of the circle and the method of determining the pattern. In the following algorithm values for variables were picked after manual testing to find the most optimal ones.

```python
# 4: Local Binary Pattern (LBP)
def fd_lbp(image, numPoints=24, radius=8, eps=1e-7):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    lbp = feature.local_binary_pattern(image, numPoints,
                                        radius, method="uniform")
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0,
                numPoints + 3), range=(0, numPoints + 2))
    hist = hist.astype("float")
    hist /= (hist.sum() + eps)
    return hist
```

5) Histogram of Oriented Gradients (HOG)

The Histogram of Oriented Gradients is mainly used for object detection and is calculated by counting occurrences of gradient orientation in localized portions of an image. For example, the image can be divided into 8×8 cells, and the histogram of oriented gradients would be computed for each cell. The final step is normalizing the histogram in order to reduce the contrast variation and improve accuracy of the descriptor. The following figure shows the HOGs for our Negative and Positive images.

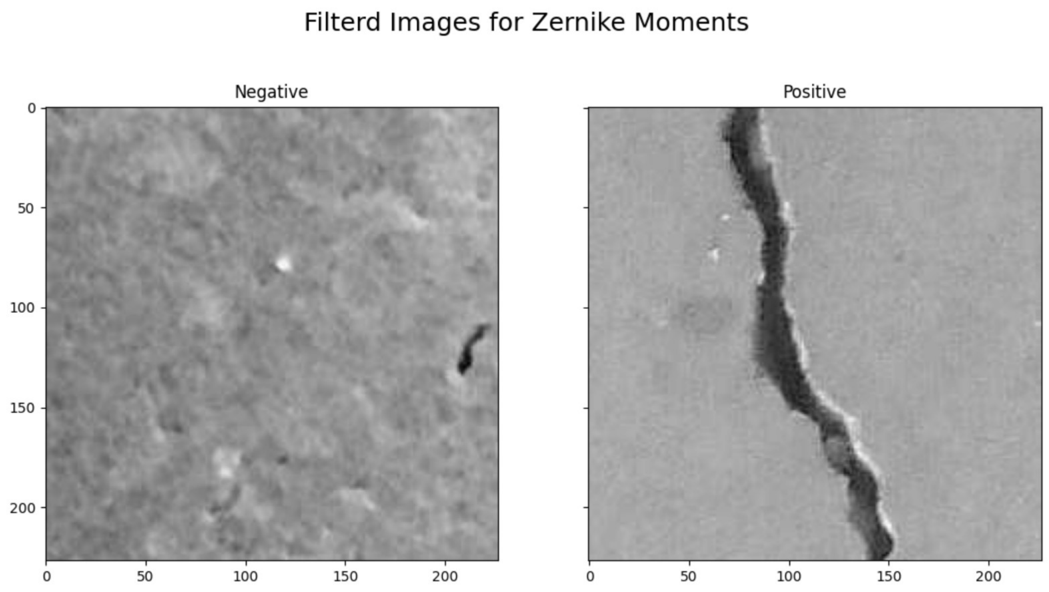


In Python, HOG is calculated using one function from skimage.feature library which returns a ready descriptor. Of course, HOG may be calculated for images with different sizes, however, the most typical convention is for the input image to have size 128 x 64 x 3 (height x width x channels) which gives the output feature vector of fixed length 3780. This also improves the performance of the program and is hence applied in the solution.

```
# 5: Histogram of Oriented Gradients (HOG)
def fd_hog(image):
    image = resize(image, (128, 64))
    fd = feature.hog(image, orientations=9, pixels_per_cell=(8, 8),
        cells_per_block=(2, 2), visualize=False, multichannel=True)
    return fd
```

6) Zernike Moments

Zernike Moments may be treated as an alternative for Hu moments. They are not as popular as Hu Moments due to their complexity which greatly impacts the algorithm execution time. Just like Hu Moments, they are a vector describing the image, however it is much longer than in case of Hu Moments. Therefore, to visualize part of the procedure, the following figure shows our 2 exemplary images after filtering out only the first channel before extracting the feature.



As for the implementation, the usage of Zernike Moments is very simple. They are calculated using a single function from the mahotas library which return a ready descriptor. As shown on the figure, the input image also needs to be slightly preprocessed - only the first of 3 channels is taken for computation. Moreover, the other two arguments: *degrees* and *radius* greatly influence the overall performance of the algorithm, so their values were chosen after manual testing, to obtain the highest possible accuracy within a reasonable amount of time.

```python
# 6: Zernike Moments
def fd_zernike(image):
    return mahotas.features.zernike_moments(image[:, :, 0],
                                             degree=20, radius=20)
```

2.2 Machine Learning Model

Since there are many classifiers to choose from, the one selected for this task was the one that gave the highest prediction score for our input data. After testing several other models, Random Forest Classifier from sklearn.ensemble library turned out to be the most suitable for the crack detection exercise. It is a supervised learning algorithm which is the most flexible and easy to use. This classifier works by constructing a multitude of decision trees (a forest) at training time, getting a prediction from each tree and then selecting the best solution by voting. Additionally, by averaging all predictions, it also corrects the decision trees' habit of overfitting to their training set.

To classify input data, the algorithm iterates over given number of photos (e.g. 2000) from both folders (Negative and Positive). Then, it extracts selected features descriptors from each image, combines them into one stack and appends this stack into global array of descriptors.

```python
# Iterate over training folders
for class_name in class_labels:
    dir = os.path.join(train_path, class_name)
    current_label = class_name
    print("\nExtracting features in folder: {} ...".format(current_label))
    for x in range(1, images_per_class+1):
        if x < 10:
            file = dir + "/0000" + str(x) + ".jpg"
        elif 9 < x < 100:
            file = dir + "/000" + str(x) + ".jpg"
        elif 99 < x < 1000:
            file = dir + "/00" + str(x) + ".jpg"
        elif 999 < x < 10000:
            file = dir + "/0" + str(x) + ".jpg"
        else:
            file = dir + "/" + str(x) + ".jpg"

        # All files are the same size so no need for resizing
        image = cv2.imread(file)
        # Feature Extraction
        fv_hu_moments = fd_hu_moments(image)
        fv_haralick = fd_haralick(image)
        fv_histogram = fd_histogram(image)
        fv_lbp = fd_lbp(image)
        fv_hog = fd_hog(image)
        fv_zernike = fd_zernike(image)

        cumulative_feature = np.hstack([fv_hu_moments,
                                        fv_haralick,fv_histogram])

        # Update the list of labels and feature vectors
        labels.append(current_label)
        features_list.append(cumulative_feature)
    print("Processed folder: {}".format(current_label))
print("Feature Extraction Completed.\n")
```

Next step consists of encoding the target labels (i.e. our classes) and normalizing all the features by scaling them to fit into range (0-1). This process ensures that labels are stored in a proper format (as unique numbers) and that any of the multiple different features used doesn't dominate others with respect to it's value. This also greatly reduces the time needed for training the classifier.

```
# Encode the target labels
encoder = LabelEncoder()
target = encoder.fit_transform(labels)
# Scale features in the range (0-1)
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled_features = scaler.fit_transform(features_list)
```

Lastly, the classifier is initiated and set of features and labels is split into training and testing part. With accordance to the task description, the testing part consists of 20% of total number of features and labels (test_size = 0.2), however this parameter can be easily changed. Then, the model trains itself (clf.fit()) and the scoring function (clf.score()) assess the accuracy of models predictions comparing them to the actual testing labels.

```
# Initiate ML model and divide dataset into training and testing part
clf = RandomForestClassifier(n_estimators=num_trees, random_state=seed)
(trainFeatures, testFeatures, trainLabels, testLabels) = train_test_split(
    np.array(rescaled_features),np.array(target),test_size=test_size,
    random_state=seed)

# Train the model and evaluate prediction score
clf.fit(trainFeatures, trainLabels)
result = clf.score(testFeatures, testLabels)

print("\nImages per class used: {}".format(images_per_class))
print("Accuracy for Random Forest Classifier: {}\n".format(result))
```

3 Results Assessment

3.1 Features Quality

As previously mentioned, the final set of features and labels is divided into 2 parts: 80% for training and the remaining 20% for testing. However, since the provided data set is really large, extracting feature descriptors from all of 40 000 photos is very time consuming and doesn't really have a huge influence on the accuracy score. Therefore, for evaluating the quality of the features and their actual performance, only small part of initial data set is used i.e. 1000 images per class, so 2000 images in total. This amount is definitely enough to spot differences between the features effectiveness and enables to save time needed for extraction. As a proof, the following table shows the prediction scores for single features analyzed on 1000 images per class versus 20 000 images per class.

FEATURE	**1000 IPC**	**20 000 IPC**
Hu Moments	91.75%	94.4%
Haralick Texture	99%	99.35%
Color Histogram	99.5%	99.3875%
LBP	95.75%	97.2125%
HOG	96.75%	98.3%
Zernike Moments	74.25%	82.65%

Table 2: Features Performance (IPC - Images Per Class)

As we can observe, the most accurate feature descriptor for our problem, turned out to be the **Color Histogram**, which as a single descriptor in a stack resulted in 99.5% of correct guesses. This may be caused by the fact that cracks in our data set are much darker (usually black) in comparison to the light (gray or beige) surface they are on, which in turn causes the histograms of Negative and Positive photos to be easily distinguishable.

The next descriptor, which proved to be almost as accurate as Color Histogram was the **Haralick Texture** with prediction score of 99%. This is not surprising, considering the fact that it also evaluates colors (and their spatial arrangements) on the image.

The results for the local feature descriptors: **Local Binary Pattern** and **Histogram of Oriented Gradients** are also on a very high and similar level of 95.75% and 96.75% respectively. The biggest downfall of these two descriptors, though, is their complexity which results in high execution time. In our case, all photos are really small, so those features performed quite well. However, the overall conclusion would be that LBP and HOG, despite their high accuracy, may not be the most efficient solutions for large data sets, especially with bigger images.

Results of shape recognition functions - **Hu Moments** and **Zernike Moments** were significantly lower than any other descriptors - 91.75% and 74.25%. Having to choose one, it would certainly be better to use Hu Moments due to its higher accuracy and much better performance. Although in this particular case Hu Moments had really high score, the above results bring forward the conclusion that shape recognition alone may not be the best solution for crack detection, especially on smaller data sets.

After evaluating the results of single feature descriptors, we can test their performance together. The following list presents the prediction scores of the few selected features combined in groups:

- Hu Moments + HOG: 98.25%
- Hu Moments + Zernike Moments: 92.75%
- Haralick Texture + Color Histogram: 99%
- Haralick Texture + HOG: 97.5%
- Color Histogram + LBP: 99.25%
- Color Histogram + Hu Moments: 99%
- Color Histogram + Zernike Moments: 99%
- LBP + HOG: 97.25%
- LBP + Zernike Moments: 94.5%
- Color histogram + LBP + HOG: 99.25%
- Haralick Texture + LBP + HOG: 98.5%
- Haralick Texture + Zernike Moments + Color Histigram: 99%
- Color histogram + Haralick Texture + Hu Moments: 99%
- Color Histogram + Haralick Texture + LBP + HOG: 99.75%
- Hu Moments + Haralick Texture + Color Histogram + LBP + HOG: 99.5%

The highest prediction score was obtained after combining 4 best feature descriptors (without Hu and Zernike Moments) which means that those two descriptors were too inaccurate and were slightly lowering the final result. Furthermore, we can observe that combining features had visible effect on the score only when one of the descriptors had noticeably (at least few percent) lower score than the others. In such case the final result was usually somewhere between the scores of single descriptors, like combining Haralick Texture + HOG - from [99%, 96.75] to 97.5%.

However, there were also cases were 2 descriptors with similar results gave much higher prediction scores when combined. For example, Hu Moments + HOG - from [91.75%, 96.75%] to 98.25% or LBP + HOG - from [95.75%, 96.75%] to 97.25%. This means those descriptors well complimented each other. The interesting thing is also the fact that after combining 2 descriptors which alone had the highest prediction scores (Haralick Texture and Color Histogram) ... which is the complete opposite to Zernike Moments. This descriptor alone had much worse results than all descriptors, but when combined with any other feature performed unexpectedly well (e.g. LBP + Zernike - from [95.75%, 74.25%] to 94.5%).

To summarize, the overall results are definitely impressive. Almost all descriptors had the prediction score above 90% which is really precise and proves that even using only one feature we are able to classify the cracks almost perfectly. However, this is not surprising considering the large size of the data set and very limited number of classes (just 2). The only feature descriptor which achieved relatively low score (in comparison to other descriptors) was Zernike Moments which shows that, in this particular task, they were not a good alternative for Hu Moments.

3.2 Impact of the Sample Size

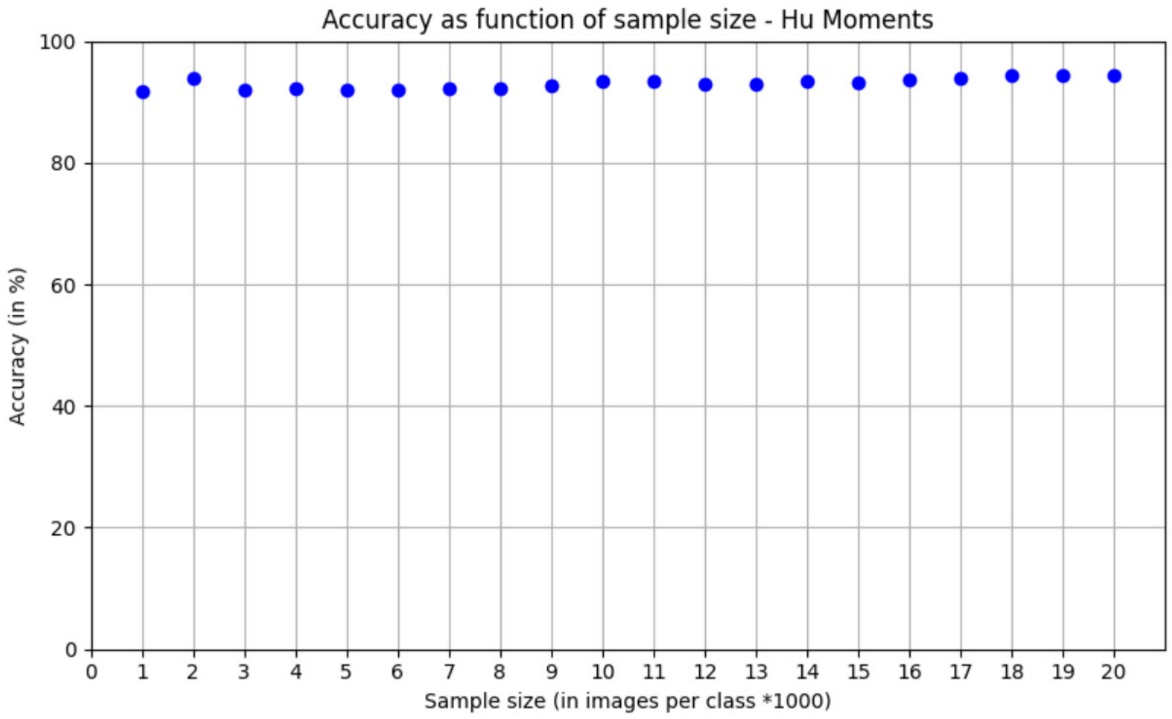
In order to assess the impact of set size on the prediction scores, the measurements were taken for the 2 least accurate descriptors: Hu Moments and Zernike Moments. This choice was made due to the fact that evaluating more than one descriptors combined on such large sample, would be very time consuming. Moreover, since the performance of 4 out of 6 descriptors on sample of only 2000 images was already almost perfect (above 95%) there wasn't much room left for the improvement and the measurements would probably turn out to be inconclusive.

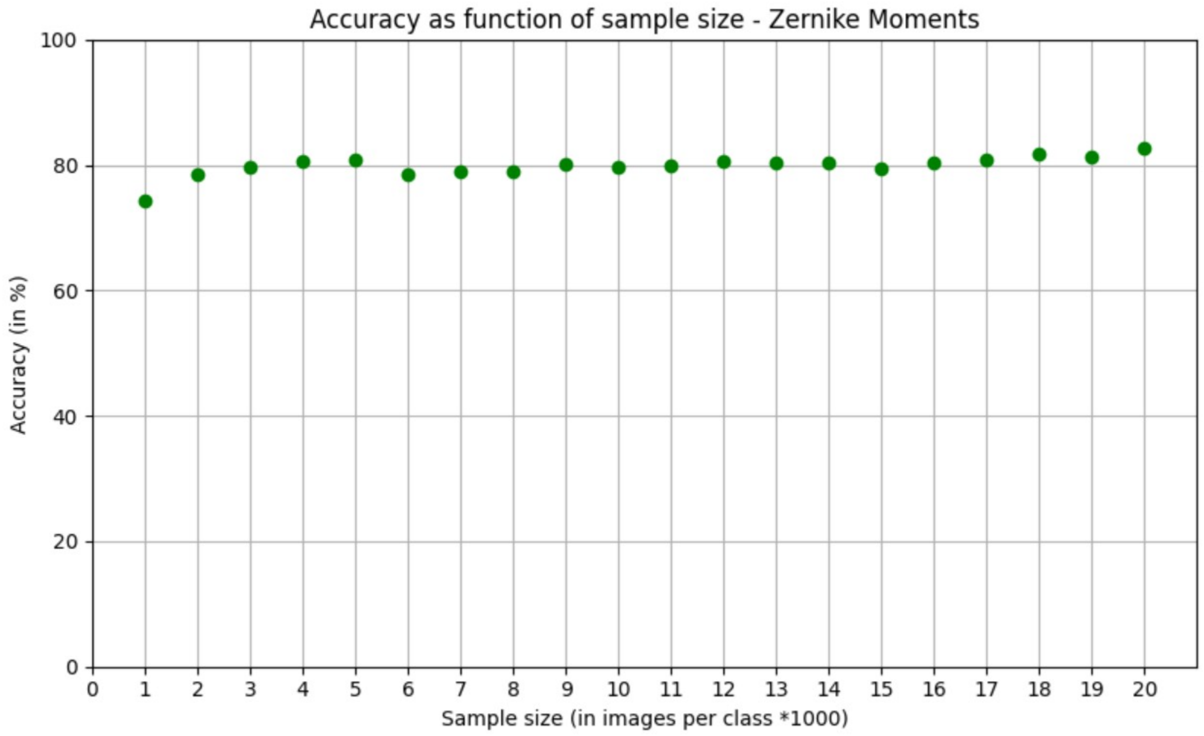
The following table shows accuracy results for Hu Moments and Zernike Moments obtained for each of the sample size - starting at 1000 images per class and increasing the number by 1000 for every measurement.

Sample size (IPC)	**Hu Moments**	**Zernike Moments**
1000	91.75%	74.25%
2000	93.875%	78.625%
3000	92%	79.58%
4000	92.31%	80.5%
5000	92%	80.75%
6000	92%	78.54%
7000	92.32%	79.07%
8000	92.28%	78.97%
9000	92.64%	80.19%
10000	93.425%	79.65%
11000	93.41%	80%
12000	92.94%	80.71%
13000	92.885%	80.44%
14000	93.375%	80.41%
15000	93.18%	79.45%
16000	93.81%	80.41%
17000	93.985%	80.93%
18000	94.46%	81.72%
19000	94.355%	81.37%
20000	94.4%	82.65%

Table 3: Feature accuracies for respective sample sizes (IPC - Images Per Class)

For better visualization, these results were also plotted on the following diagrams. The sample size is given in 10^3 images per class.


Accuracy as function of sample size - Hu Moments
Accuracy (in %)
Sample size (in images per class *1000)


Accuracy as function of sample size - Zernike Moments
Accuracy (in %)
Sample size (in images per class *1000)

As we can observe, the impact of the number of images used for classification on the prediction results is not as large as one may have expected. In case of Hu Moments, the accuracy improved by only $\sim$3% and the relation between accuracy and sample size is more constant than linear. The growth of the function is really slow, considering the fact we significantly increase the number of photos for every measurement. Moreover, the score for 2000 images per class turned out to be as high as for 16 000 images even though next 7 results are on the level of $\sim$92%. Such deviations may be caused by the differences in photos used for feature extraction.

As for the Zernike Moments, the improvement of accuracy is much more visible - around 8%. However, the relation is also not quite linear. The highest improvement in prediction score can be observed for first 5 measurements. After that the accuracy is rather constant and waver around $\sim$80%, just to slightly increase at the end, where it reached its maximum of 82.65% for 20 000 images per class.

In conclusion, the influence of sample size on the accuracy scores is not that significant. This outcome is probably due to very large set of images for only 2 classes to distinguish. Therefore, using even small part of initial data set for feature extraction and model training is enough to achieve almost perfect results, which are hard to improve by just providing more images.

4 Additional Notes

The solution presented in the report can be found in the source file classify.py. The additional file named visualize.py is only for visualization and after running, will display all the figures presented in the report. The assumed structure of folders is as follows: folder "Data" containing folders with Names of labels to classify - in this case "Positive"and "Negative" folders.

References

[1] https://learnopencv.com/shape-matching-using-hu-moments-c-python/

[2] https://www.pyimagesearch.com/2014/01/22/clever-girl-a-guide-to-utilizing-color-histograms-for-computer-vision-and-image-search-engines/

[3] https://www.pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/

[4] https://analyticsindiamag.com/image-feature-extraction-using-scikit-image-a-hands-on-guide/

[5] https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/?utm_source=blog&utm_medium=3-techniques-extract-features-from-image-data-machine-learning

[6] https://www.analyticsvidhya.com/blog/2019/08/3-techniques-extract-features-from-image-data-machine-learning-python/

[7] https://www.datacamp.com/community/tutorials/random-forests-classifier-python